An Introduction to
Component Technology
and Middleware

by

Matthew Wilkins
M.Sc., Courant Institute, New York University, 2000

AN ESSAY SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this essay as conforming
to the required standard

The University of British Columbia
April 2002
© Matthew Wilkins, 2002
Abstract

Component technology is introduced and three major technologies: CORBA, EJB, and COM+ are discussed. The aspects covered include language and platform independance, performance, security and reliability. The general fate of component technology is also presented.
Contents

Abstract ii

Contents iii

Acknowledgements v

1 Introduction 1

2 Component Technologies 4

2.1 The Open Management Group 4

2.2 Sun Microsystems 8

2.3 Microsoft 12

2.3.1 Object Linking and Embedding 13

2.3.2 Component Object Model 13

2.3.3 Distributed Component Object Model 14

2.3.4 Microsoft Transaction Server 15

2.3.5 Microsoft Message Queue Server 16

2.3.6 COM+ 17

3 Technology Comparison 19

3.1 Language support 19
Acknowledgements

You can thank Norm Hutchinson for the subject of this essay, I certainly do, for I found it most interesting learning about Component Technology. Also thank you Norm for the time you spent with me and the interesting discussions we had.

Thanks goes to Heather Jenkins for proofreading not only this essay but most of my other reports and essays throughout my Master’s studies.

MATTHEW WILKINS

The University of British Columbia
April 2002
Chapter 1

Introduction

The stand-alone computer concept has been dead for quite sometime now. Everyone is, or wants to be, “connected”. Users want to be able to retrieve information and to interact with systems anywhere. A computer’s sole value used to be in and of itself, but now the value of a computer is increasingly thought to be in what it can provide in terms of information or services from elsewhere. A computer as a glorified typewriter is not very fun; people want to play online games, interact with other computer users, and surf the web. Businesses want consumers logging on and buying merchandise from them over the net.

Corporations are frequently declaring that their information is an asset, perhaps their most important asset. But to make use of this information effectively it must be accurate and accessible everywhere. This means machines must be connected together and be able to freely pass information back and forth. However there are many hurdles. Computers are so complicated and varied. It is likely that the very machines that need to communicate are running different operating systems, using different protocols, in different countries, or owned by different people with diverse operating principals and polices.
Individual projects within organisations want the freedom of selecting tools to best suit their needs. Also many different sorts of applications exist, ranging from custom mainframe solutions to client/server applications. It is beneficial to integrate all these disparate systems because it enables software sharing and reuse, but without a comprehensive architecture to mask all the differences across projects and applications it is difficult.

Object orientated programming was touted as the solution to all programming problems. However, even using object orientated technology, it is still not easy to write big complex pieces of software. Object orientated programming has definitely helped matters but not enough. Software development costs are not decreasing like hardware costs are. The time to develop, maintain, and enhance software places limits on how big a system can practically grow.

One solution to the above four general problems is Component technology. Components are reusable building blocks that can be combined with other components to form applications. In practice a component is often just a big object, made up of smaller objects, that provides some sort of service. Components increase the size of system that can be managed by increasing the size of the basic building block from objects to components. Components are supposed to be able to communicate with each other over heterogeneous networks. These two features help with writing large software applications, and help programmes communicate with each other regardless of platform, age or location. Writing distributed applications is easier with components. Getting information from place to place is easier.

A key tool in component technology is the interface. The interface represents components in the heterogeneous distributed environment in a standard way so all components can communicate. With interfaces come interface definition languages
which are used to define them.

One of the goals, perhaps the most important one, of component technology is interoperability, so it is unfortunate that there are a few competing standards. I will discuss the Common Object Request Broker Architecture in Section 2.1; the Java solution, Enterprise Java Beans, in Section 2.2; and the offering from Microsoft in Section 2.3. The last two technologies have middleware elements to them, too. Middleware is something at a higher level than component technology and aims to provide more services to components in a seamless fashion.
Chapter 2

Component Technologies

This chapter gives some history and an introduction to the three players in component technology that will be discussed in this essay: the Open Management Group, Sun Microsystems, and Microsoft.

2.1 The Open Management Group

The Common Object Request Broker Architecture (CORBA) version 1.1 specification was introduced by the Object Management Group (OMG) in 1991 [10]. The OMG is an international consortium of over 800 members consisting of vendors big and small, software developers, and users. It was founded in 1989 to promote object orientated technology in software development. The OMG’s mission includes the establishment of guidelines and specifications to provide a common framework for application development. The base level of this framework is the Object Management Architecture (OMA) which is a “conceptual infrastructure” [5]. CORBA is based upon the OMA, as are many of the specifications from the OMG. CORBA is probably OMG’s best known work.
Another important specification from the OMG is the OMG Interface Definition Language (OMG IDL) adopted in 1991 [10, chapter 3]. This specification is the primary tool that CORBA uses to insulate components written in different languages from each other. The OMG IDL is simply a language independent way to define interfaces. It includes the most basic types like integers and doubles, and more complicated ones such as unions and arrays. Other features are: contextual information, as in the UNIX environment; references; whether parameters are in, out, or both; exceptions; execution semantics, for example at-most-once; and whether the method is expected to return. The OMG IDL is not a fully fledged language but is broad enough for its purpose and mappings for many languages exist.

The CORBA version 2.0 specification, formally adopted in 1997 [4] was designed to allow developers to create components that can interact with other components without requiring any knowledge of how or where that component was implemented. The “where” implies that CORBA is location transparent, there is no way for a client to know where the server is running. The “how” implies that CORBA supports components in a heterogeneous environment, components may be running on an Intel Windows machine or a Sparc running Solaris, for instance. Each CORBA component is represented by an interface with an associated set of methods, and clients hold references to components. There is multiple inheritance with the root super classes being CORBA::object and the Basic Object Adaptor class which is created by the compiler for each class to map between interface and implementation.

Figure 2.1 gives the architectural model of CORBA version 2. The heart of the system is the Object Request Broker (ORB) core, or the “object bus”. It is through this that clients make requests to servers. The ORB is responsible
for finding the implementation for the request, getting the implementation running, and handling all the communication between client and server. From looking at Figure 2.1 one can see that the ORB does get a lot of help from other things though. The protocol the ORB uses to connect the distributed components is the **Internet Inter-ORB Protocol** (IIOP). IIOP is a specialisation to TCP/IP of the message formats and data representations of the more general **General Inter-ORB Protocol** (GIOP). The ORB Interface allows both clients and servers to talk directly to the ORB for common tasks such as checking existence of components, duplicating references, converting references to strings, etc.

The **Object Adaptor** (OA) is the primary way a server interacts with the ORB. It is also the way the ORB talks to the server component. The OA is responsible for things such as generating component references, method invocation, security, and activation/deactivation. The last task is important for performance
reasons. A heavily used machine may have a lot of components, many of which could be idle, especially if there is a person involved at the client end. Being able to deactivate (release held resources and so on) such components is useful. The programmer of the server component can set resource patterns in the OA to control the way the component is handled. The mechanism is very flexible indeed but simple polices such as session and entity exist for convenience. The session policy is used for a component that represents some sort of action, it is short lived and many instances can exist. The entity policy is used for a component that represents something, like a row in a database. It is persistent and unique.

There are two ways to invoke a service from a component: statically and dynamically. The static skeleton is generally an up-call like interface to a component's implementation and can be compiled right into the server. The static stub is the counterpart, it is compiled into the client and manages the client language dependent-ORB meeting point. This stub/skeleton arrangement is similar to what Sun Remote Procedure Call (RPC) does [18]. The dynamic stub (sometimes called the Dynamic Invocation Interface) allows for calling of methods unknown at compile time. The ORB can store interface information in a special repository that the client can access at runtime, and it can then use the dynamic stub to make the actual call. On the server side is the dynamic skeleton which is used for methods that have not been embodied in a static skeleton or are being called by a client using a dynamic stub. Using a dynamic skeleton means servers do not have to publish their services until runtime. Also the dynamic stub/skeleton architecture can be used to construct inter-ORB bridges that are used to connect different ORBs (perhaps from different vendors).

The latest version of CORBA, version 3, has not been officially adopted yet,
although it has been in the works for a few years [22]. One of the major additions is a “middleware” layer based on Enterprise Java Beans (EJB). These beans will be discussed more in the following section. Unless otherwise stated in the rest of this essay CORBA will refer to version 2 without the beans.

2.2 Sun Microsystems

In 1995 Sun introduced its Java technology, and the following year the Java Development Kit version 1.0 appeared as a free download [2]. Java’s initial popularity was mainly due to its support for downloadable Java programmes called applets. Later Sun introduced the Java Servlet Specification. Servlets are in many ways analogous to applets except they run on the web server. They are often used to generate Dynamic web content. General purpose Java client/server communication has been in Java since the 1.1 release and goes by the name of Java Remote Method Invocation (Java RMI).

Java RMI uses the Java Remote Method Protocol (JRMP) which relies on object serialisation. Serialisation is the process by which an object is converted to a bunch of bits so it can be transmitted as a stream. Communication to and from the server is handled in this fashion, so very complicated objects, not just basic types, can easily be used as arguments or return types. This is how exceptions are thrown too, as they are just objects. Since serialisation and the JRMP are particular to Java, both client and server must be written in Java. One of the pleasant things about Java is its garbage collection, and this has been extended to handle the distributed case, through the use of leases and timeouts [24].

Servers implement an interface which describes the server’s methods to clients. This interface extends the java.rmi.Remote interface. The server then registers itself
with a well known machine, and the client can then query that machine to get a reference to the interface. With this, the client can make calls to the server. A server does not have to publish all its methods though, because the client is able to discover them using reflection.

Despite the sophistication of Java RMI it is not really a component architecture, rather it is the Java equivalent to Sun RPC [18], albeit a far more convenient one.

In 1997 Sun announced its support of the CORBA specification [3]. This means that objects using RMI to communicate do not have to use the JRMP, rather they can use the IIOP. Java servers can serve any CORBA client, and a Java client can invoke methods on a CORBA server. This, along with CORBA’s active incorporation of Enterprise Java Beans (see Section 2.1), has tied Java and CORBA tightly together.

Combining CORBA and RMI brought component technology to Java but did not create a “middleware” layer, that came when Sun announced (also in 1997) its Java 2 Platform Enterprise Edition (J2EE) initiative. The J2EE basically consists of a bunch of Enterprise Java application programming interfaces (API), the cornerstone being the Enterprise Java Beans (EJB) API. EJB defines the server-side component model. It is not the only thing in J2EE, other technologies include:

- Java Interface Definition Language
- Java Message Service
- Java Naming and Directory Interface (JNDI)
- Java RMI and Object Serialisation

9
- Java Servlet API

- **Java Transaction API** (JTA)

- Java Transaction Service

- Java Server Pages

- JDBC (often incorrectly thought of standing for Java DataBase Connectivity)

The technologies are inter-related, for instance EJB depends on JNDI for locating components and on JTA for handling transactions, but I will mainly concentrate on EJB in this essay since it is the major part of Java’s (and since version 3.0, CORBA’s) middleware component technology.

The EJB version 1.0 specification was released in March 1998 [17], and its architecture (see Figure 2.2) can be broken into the pieces:

![Figure 2.2: The Enterprise Java Bean architecture.](image)

- The client. This could be an application running on a user's PC, or perhaps a Java Servlet running on a server providing dynamic webpages. The client
can be written in any language and uses IIOP for communication (or it can use RMI if it is a Java client). The client gets a reference to the EJBHome object, using JNDI for instance, and then uses that to create a reference to an EJBObject object. This can then be used to access the bean’s services.

- The application server provides all the underlying services required by the bean. These include: transactions, naming, database access, security, life cycle and thread management.

- The container is quite closely tied to the application server. It provides the environment in which the bean runs, and provides the application server’s functionality to the bean. In addition it helps with registering beans, creation and deletion of bean instances, making beans persistent and authenticating clients since it can look at the bean’s access control lists.

- The bean itself is where all the “logic” goes. There are two types of bean: Session and Entity. Session beans represent an action that the server can perform. There can be many instances of this bean, some for each client if needed. They can be stateless or stateful, where the bean remembers its state between calls. Entity beans represent something, like a row in a database, and are unique even in a distributed setting. They have a “key” associated with them to identify the data within them. The container can manage the Entity bean’s database access for it, or the bean can do it itself which gives more flexibility and is useful for legacy data access.

The classes that represent the two interfaces EJBHome and EJBObject are created automatically when the bean is loaded into the container. The EJBHome class contains a create method for each of the bean constructors, each taking
a different set of parameters. For entity beans there is also a “finder” method which takes a key and returns the EJBObj ect. The EJBObj ect is a proxy the client can use to access the bean’s methods.

As mentioned at the end of Section 2.1, EJB and CORBA are now tied together. On the 6th of April 1999 the OMG announced that the EJB model would be used as the core of CORBA’s middleware layer [20]. Because EJB and CORBA have not in practice assimilated each other, in the rest of this essay EJB will generally refer to just the EJB specification and not the CORBA/EJB combination.

2.3 Microsoft

Microsoft (MS) offered its latest middleware solution in 2000 with the introduction of its .NET framework. The .NET solution was much trumpeted and meant different things to different people. In reality .NET is not one thing, rather a collection of the latest incremental improvements in various MS products all rolled together with a few new things thrown in. Here is a list of .NET components [8, 21]:

- C#. MS’s Java with metadata tags so a class can be automatically treated as a component.

- Runtime environment which includes:
  - MS Internal Language (MSIL) which is like JavaByte code
  - Common language runtime and compiler which runs the MSIL
  - Component orientated middle-tier infrastructure, COM+

- OS libraries packaged up as components
• ASP+ or ASP.NET, a new version of Active Server Pages (ASP) that is compiled rather than interpreted (by virtue of running in the MSIL which ends up being compiled)

• ADO+ or ADO.NET, a new version of Active Data Object (ADO) data access that uses XML and Simple Object Access Protocol (SOAP) for data exchange.

• WinForms and WebForms, new user interface components for web applications and desktop applications.

Since this paper is about component technologies and middleware, out of all the various aspects of .NET, COM+ will be the one focused on from now on.

2.3.1 Object Linking and Embedding

It started with MS’s Object Linking and Embedding (OLE) proprietary document and integration framework for MS Office in 1991. This enabled users to embed spreadsheets in MS Word files etc. In 1995 MS realised that OLE was a special case of component communication and integration and released COM which provided general-purpose component integration.

2.3.2 Component Object Model

COM is a binary compatibility specification and implementation that allows clients to invoke methods on COM objects or components. Components have to lay out virtual function tables (vtables) in memory in a standard way [23]. Then any language that can call functions via pointers (that is, most languages) can be used to write components. Through double indirection, vtables can be shared.
At the source code level, COM objects expose their services through interfaces defined in the MS Interface Definition Language (MIDL). The interface is a strongly-typed contract between components to what services a component provides. A client can only have a reference to an interface, not the object implementing it. Multiple inheritance is allowed and in fact this aids in version control. Every interface has its own interface identifier, a globally unique ID, and it cannot change. There are no new versions of an interface. When new functionality is required a whole new interface is made and a component is written to implement it (perhaps the same one as was implementing the old version can have a few methods added). Clients hold references to the interface and when they call methods, control goes to the correct implementation. It is possible for a client to dynamically query a server as to what interfaces it implements. This means components do not need to publish their interfaces till runtime.

COM objects run inside Dynamically Linked Libraries or normal executable EXEs. Error handling is rudimentary, being handled using return codes. All components must implement the interface IUnknown, which amongst other things is responsible for handling reference counting. This means components must keep track of who is using them so they know when they can unload themselves. There is no distribution, client and server run on the same machine although they can run in different address spaces. COM objects can be written in any language that can be compiled to the necessary binary standard and be run on a MS OS.

2.3.3 Distributed Component Object Model

Distributed COM (DCOM) arrived in 1996. It added networking support by providing RPC based on the Distributed Computing Environment (DCE) RPC.
If the client and server are on the same machine then the normal COM mechanisms are used, else, an RPC is made to the COM run-time on the server’s machine and then that takes over. So DCOM is essentially a bridge between COM run-times.

DCOM supports a variety of transport protocols including TCP/IP, IPX, and NetBIOS [9], but most commonly it uses UDP. Requests are batched together which is a very important optimisation since network times are often very high (milliseconds, perhaps up to a second). It does support a sort of directory service whereby a client can contact a central location for a service only to receive back a reference to a component on another machine. This can be used for primitive load balancing, too.

Security is whatever the underlying OS can provide, so in the case of DCOM on Windows this means Windows NT (or Windows 2000) security. DCOM adds RPC to COM but that is basically it. As explained in MS’s DCOM Technical Overview [9], DCOM provides the framework to build scalable, fault tolerant etc. systems but does not provide those features itself.

2.3.4 Microsoft Transaction Server

Also in 1996 came the introduction of the **MS Transaction Server** (MTS). It was designed to simplify the development of the middle tier by providing infrastructure for the developer. MTS is a layer on top of DCOM that adds transaction support. To be precise, MTS interposes itself between DCOM components by “munging” the registry on the server [13] to provide the following three services: a thread pool, a database connection pool, and a transaction monitor. All other aspects are as in DCOM, for instance, the security model for MTS is whatever DCOM can provide.

The thread pool management relieves the developer from having to deal with
threads or processes. When a request for a service is intercepted the MTS will get
a thread from its pool to run the service, returning the thread to the pool when it
is finished running the component method.

The database connection pool is a real performance boost. It takes a long
time to set up database connections, it is also a burden for the developer to manage
their own connections. With the connection pool a developer can just write SQL
queries in the component, and MTS will assign a database connection to field them.

As the name indicates, transaction support is the major part of MTS. Trans-
actions may access a single resource manager, or with the support of the **MS Dis-
tributed Transaction Coordinator** (DTC) multiple distributed resource man-
egers. It is fairly easy for the developer to use transactions, all one has to do is set
a property in the component and MTS will take care of the rest.

### 2.3.5 Microsoft Message Queue Server

In 1997 the **MS Message Queue Server** (MSMQ) was released [6]. It is based
upon the open **Message Queue Middleware** (MQM) standard. The MSMQ
allows distributed components to communicate asynchronously in a reliable way over
unreliable networks. It is no good for tightly coupled synchronous communication
where one waits for a response back. It is more useful in a “news” type situation
where the server propagating messages does not really care when they arrive.

The communication is connectionless, and messages can be stored on inter-
mediate machines until network connectivity resumes. The most stringent semantics
that can be guaranteed are at-most-once or at-least-once delivery. MSMQ is use-
ful for the type of situation where message delivery times are not important, for a
message may take seconds, minutes, hours or even days to make it.
2.3.6 COM+

Finally COM+ was introduced in 1999. This combines all the above into one package, adds a few extra things, and promised some more. The promises were load balancing and in memory databases which did not eventuate. The things added were mainly to make routine tasks easier or nonexistent for the developer. There are lots of annoying things about COM ranging from the obvious such as having to manually keep track of references to control one's lifetime, to the less obvious, like having to manage the interface definitions keeping them synchronised with the source code and so on. COM+ rolls everything into one package and attempts to tidy lots of these things up to make it easier to use.

To be a bit more precise, COM+ makes it easier to write COM components by suppling: default class factories and IUnknown interfaces, registration code, and automatic reference counting. A lot of COM code can be fairly similar, moving it into the COM+ runtime reduces developer effort.

The second thing COM+ does is help with the IDL, in fact one does not need to define interfaces using MIDL. Rather, interfaces are defined using whatever COM+ enhanced programming language the developer is using. Tools in COM+ can build the proxies and stubs automatically.

The third change is that COM+ provides a bunch of new features: objects can be directly exposed so clients can reference an object not just its interface, implementation inheritance not just interface inheritance, garbage collection, and a new security model based on roles and privileges. The first two features give power but also problems, and the garbage collection is because .NET's language of choice, which is C#, is automatically garbage collected. Components are assigned a certain privilege required to use them and users have a privilege assigned by the system.
administrator. These must match for access. Also the user must be playing the
correct role to access the component [13].

The fourth and final thing introduced by COM+ is interception. Recall that
MTS basically intercepts clients calls to components to handle transactions, which
is fine and dandy until one wants to add another interception service. It would
interfere with MTS and vice versa. COM+ introduces a general mechanism for
defining interceptors.
Chapter 3

Technology Comparison

Now that the technologies have been described individually, this chapter can compare them on specific issues.

3.1 Language support

EJB is really only a Java solution. Beans need to be compiled to JavaByte codes. However this does not necessarily mean that the Java language has to be used, compilers for other languages such as lisp and perl have been developed [15], although their use does not seem to have caught on. As of CORBA version 3, EJB and CORBA are tightly coupled (see Section 2.2). There is a reversible mapping between OMG IDL and Java, so interfaces written in Java and interfaces in OMG IDL are interchangeable. Also EJBs now support IIOP. This means CORBA clients, written in any language CORBA supports, can communicate with any EJB.

CORBA was defined from its initial inception to be language independent. Its mechanism for being so is the OMG IDL. The developer needs to learn the OMG IDL to write interfaces in and then apply a stub/skeleton generator for their
language of choice. The stubs and skeletons are in the chosen language and provide all the necessary code to marshal data into and from the ORB. Lots of languages are supported including C, C++, Ada, and Java.

DCOM is similar to CORBA in the fact that there is an IDL, MIDL in this case, and a DCOM language must be able to map to the MIDL. However compatibility is really defined at the binary level, all components must follow the binary standard vtable function pointer layout. Supported languages include C, C++, VisualBasic, COBOL and probably more.

COM+ mixes all the above approaches. It is like EJB in the sense that code is compiled to an intermediary language (MSIL in this case). There are interfaces, although the developer need not write any. Finally it is like COM because the MSIL is often ultimately compiled into binary code that conforms to the COM binary vtable standard.

A fully enabled .NET language must provide three things. Firstly a compiler that can convert the language to MSIL plus metadata, which is essentially the interface. This means the developer need never learn how to write interfaces in MIDL. Secondly a tool that is able to reverse this process must exist. The tool should be able to take any MSIL plus metadata and present a view of it in the language. This means the developer does not have to be able to mentally parse an interface written in MIDL. Finally the fully enabled .NET language must allow subclassing of components regardless of what original language was used to create them. Supported languages include C++, COBOL, VisualBasic and C#, with C# being the most favoured and therefore probably the more comfortable and well integrated.

COM+ is the clear winner when it comes to language interoperability. MS
has made an effort not to disenfranchise any programmers from its .NET framework.

3.2 Platform

COM, DCOM and COM+ are really Windows only solutions. MS paid Software AG
to port DCOM to various Unixes [11] but the ports were never fully functional or
 compatible. One reason could have been the lack of a proper definition for DCOM,
see Section 3.3. MS will probably not port COM+ to other platforms, and in fact it
may not be possible. One difficulty is the lack of a definition for COM+, there is no
official standard. Another problem is that COM+ is tied so deeply into other MS
products including Windows 2000 itself that such a port would involve porting key
elements of Windows too. Also, it may not be in MS best business interests. One
piece of COM+ that has the potential to be more portable than the rest of COM+
is MSMQ since it is based on the open standard MQM. IBM’s MQ (WebSphere MQ
family) runs on many different platforms [14], and since MSMQ interoperates with
it, MSMQ can indirectly work with multiple platforms.

CORBA is platform independent since it is a standard not a product. As far
as products go, more than 30 platforms are supported [11], in fact more MS OSEs
are supported than DCOM supports. However this support is provided by different
companies which does create problems. Different vendors add different functionality
to their ORBs resulting in incompatibilities. The CORBA standard is rather loose
too, which means ORBs from different vendors may not work together correctly.

EJB is supported on any platform Java is supported, in other words most
platforms. One problem EJB does have though is that the interfaces between the
bean and container are not defined properly and vendors have added extra features.
This means that the theoretically possible action of mixing and matching beans and
containers may not be possible in practice.

It is clear that COM+ is the least platform independent, but it is hard to say which is better out of CORBA and EJB. Although it probably does not matter too much which is better since both are beginning to merge taking on the weaknesses and strengths of each other.

3.3 Standards

MS has not published standards to define COM+, rather, COM+ is defined by an implementation. CORBA and EJB are published standards. However this is not necessarily a strength of CORBA/EJB and a weakness of COM+.

Standards are important and without a standard it is hard to have interoperability. One of the reasons COM+ is not portable to other platforms is due to the lack of a standard, see Section 3.2. Also the developer is often faced with poor documentation and not even the official standard to fall back on.

On the other hand, written published standards do not run on machines, they do need to be implemented. This process is difficult if the standard is not precise, and it isn’t in the CORBA case. The CORBA specification is very long too, making it difficult for it to be complete and accurate. The result is that the CORBA and EJB implementations are not perfectly compatible or “standard”.

The best situation is to have a standard and a reference implementation. But in this particular case, COM+ is only an implementation and CORBA/EJB is only a standard. So the choice is between either an implementation which cannot be ported and is at times hard to work with due to its vagueries, or many implementations that are easier to work with but are only sort of compatible with each other. At least with CORBA and EJB it is possible for many vendors to provide products,
which results in more choice for users.

3.4 Wire Protocols

DCOM uses its own UDP based protocol, or the COM protocol if the communication is within one machine. COM+ can use the DCOM protocol or a new standard, SOAP/XML, for communication [7]. SOAP is a simple wire level standard (not official but has been submitted to the IETF) for object communication that uses XML. CORBA uses IIOP, and EJB can use IIOP or JRMP. There do exist COM-CORBA bridges so in theory any of the technologies should be able to talk to any of the others.

None of the technologies are really better than any of the others; COM+, CORBA, and EJB are all able to use an open standard. However the wire protocol of choice for COM+ is probably DCOM, not SOAP, due to the relative newness of SOAP and MS’s history of using DCOM.

3.5 Performance

There are many performance related issues ranging from efficiency, such as method calls per second for instance, to scalability or load balancing, which could be thought of as a reliability characteristic, too.

Comparing the efficiency of the COM+ implementation to the efficiency of the CORBA or EJB specification is not a valid comparison. One must compare implementations. However there are many CORBA and EJB implementations to choose from, all with different performance characteristics. Tests have been done [16] that show GNOME’s ORBit is 10 times faster than Mico, but ORBit hardly manages
to fulfil the CORBA specification and Mico provides other benefits and features. Also one cannot measure the efficiency of COM+ without measuring the efficiency of Windows itself.

None of DCOM, COM+, CORBA, or EJB provide much in terms of scalability or load balancing. The only real support is the obvious fact that components can be distributed over a group of machines to perhaps increase performance or reliability. COM+ promised to bring load balancing and in memory object caching for clients, but these features failed to appear. Ad hoc solutions are available of course, but they are not integrated into the .NET framework. The OMG’s standard wire protocol, GIOP, does allow re-direction for load balancing [19] and many vendors have included load balancing in their products, but it has not been standardised.

In summary, none of the technologies address performance related issues in a consistent manner. The CORBA and EJB specifications rely on vendors adding the extra features to their implementations, and MS relies on other software companies to provide the extra features that COM+ does not provide.

3.6 Security

DCOM and COM+ rely on the security mechanisms of the underlying OS, in other words, the security provided by Windows. Under restricted conditions Windows NT security was rated to level C2 by the National Computer Security Center [1]. Windows 2000 added the concepts of roles and privileges to the Windows NT security model, see section 2.3.6 for more details.

The CORBA security specification is very comprehensive, it is 434 pages long [12]. However the ORB products are less comprehensive. Only recently have Visigenic partnered with MITRE to deliver the first ORB to comply with CORBA
level 2 security [11]. In the defence of CORBA, its security measures must be far more general and broad than DCOM/COM+ because of the heterogeneous nature of the specification and intended environment.

EJB is similar to DCOM or COM+ in the sense that it relies upon something else, the Java security model in this case, to enforce security. This model interfaces well with the CORBA security specification, and is easy for the developer to set up. Security rules are defined for each bean and the actual checks are made on behalf of the bean by the EJB container.

The loser here is CORBA. It may have a comprehensive security model, but cannot be very implementable if only one vendor supports it. Both COM+ and EJB have good security models on paper and which provides a more secure environment probably only depends on the implementations’ robustness.

### 3.7 Transactions

Transparent transaction support is a key characteristic of middleware solutions. DCOM and CORBA are not middleware solutions and do not support transactions. Both EJB and COM+, being middleware solutions, do support them. EJB transaction support is based upon the OMG **Object Transaction Service** (OTS) specification, and COM+ uses MTS.

A COM+ component specifies its transactional requirements and the MTS handles the details. A component may specify that it does not support transactions, the client must handle the transaction, it can support a transaction, or it needs a new transaction on every method call. When a client calls the server, the MTS transaction client interposes itself and makes the appropriate calls to the server on the client’s behalf. The transaction client is composed of the MTS Executive and
Dispense Manager and takes on all the normal client responsibility in a transaction: creating transactions, maintaining the transaction context, instructing the transaction manager to commit or abort and so on. The second part of the MTS model is the **MS Distributed Transaction Coordinator** (MS DTC) which handles the 2 phase commit process.

EJB uses the **Java Transaction Service** (JTS) and the **Java Transaction API** (JTA). The JTA is simply the exceptions and interfaces that the client sees, and the JTS is the interface to the underlying transaction service provider. This transaction provider is normally an OTS, because the JTS interface is particularly tailored to OTS. The OTS is comprised of five parts: terminator, control, synchronisation, coordinator, and resource manager.

To summarise, the client, component, or bean, do not need to actually concern themselves with the transaction details, these are left to the container or runtime. Overall, transaction support in COM+ and EJB is remarkably similar. In fact, COM+ proponents often criticise EJB for copying the MTS transaction architecture.

### 3.8 Reliability

In a distributed computing environment where these technologies are designed to be used, reliability becomes a big issue. The probability of a failure somewhere in the system increases exponentially with the number of machines in the system. Network failures occur too. Fault tolerance is difficult to obtain and none of the technologies achieve it.

DCOM has a pinging mechanism whereby network or component failures can be detected. Due to piggy-backing it does not use an excessive amount of band-
width, but failures can take a few minutes to be detected. DCOM provides limited automatic reconnection and redirection to other servers, although the network and servers need to be set up very carefully [9]. Also there is no support for consistency or loss of information which are key aspects of any fault tolerant solution.

CORBA version 3 includes a fault tolerance specification [22], but no vendors are supporting it yet. However, Visigenic’s CORBA implementation, VisiBroker, already provides an ad hoc solution [11].

CORBA’s new specification is promising. But currently, fault tolerance support is similar to load balancing support and other performance related issues (see Section 3.5). CORBA relies on vendors adding the extra features, and MS relies on other products. One disadvantage COM+ has, is that it can only run on Windows boxes, thus reducing users choice if they attempt to gain a fault tolerant distributed system by buying big expensive fault tolerant machines. EJB and CORBA can run on many platforms including those that are often thought to be very reliable such as mainframes.
Chapter 4

Conclusions

This essay discussed three technologies: CORBA (version 2), EJB, and COM+. Often DCOM was discussed at length too, the reasons being that it is a subset of COM+ and it is valuable to understand COM+’s parts individually so as to help understand COM+ as a whole, and to have something to compare CORBA with. However MS’s component technology today is COM+. A developer starting a new project will use the whole .NET package not DCOM. This leaves us with nothing to compare CORBA with though, since it is clearly out classed by COM+ or EJB.

The OMG knew this too. Their CORBA was a generation behind MS and dying. Hence the merger with EJB in version 3 of CORBA. Although it is not an official standard yet, it likely will be soon. Sun welcomed the CORBA merge too. One criticism of EJB is it is a Java only solution. CORBA remedies this. So now it really is a two horse race: CORBA/EJB verses COM+. Or at the higher level that I am not really discussing in this essay: J2EE verses .NET.

CORBA/EJB and COM+ are quite different, yet just based on a feature list they are similar. Both support transactions, are fairly secure, do not really support load balancing or fancy performance features like fail-over. The differences lie else-
where. The biggest being that CORBA/EJB is a collection of open specifications and standards, and COM+ is a propriety implementation. Standards have to be turned into implementations by someone, propriety implementations do not, but they destroy the goal of component technology since they cannot be ported to or interoperate with every other platform.

The choice between CORBA/EJB and COM+ is really quite simple: if the domain of deployment is one consisting of MS platforms then COM+ is the way to go. If there is a non MS machine in the mix, then one better hope that there is a CORBA/EJB product to satisfy your needs.

Running CORBA/EJB on MS machines is a silly idea. MS makes it as easy as possible to use .NET and as difficult as politically possible to run CORBA/EJB. Also .NET is truly a good framework to work in when on a MS platform. MS have tried to make it easy for the developer, it is fairly language independent, supports as many features as CORBA/EJB, and one does not need to fiddle around with interfaces which is a feature not to be trivialised. Its only downfall is that it’s a MS only solution. If one has not bought into the MS world then the only choice is CORBA/EJB. Again this is not to say that CORBA/EJB is inferior. It is platform neutral, sort of vendor neutral, supports as many features as COM+, and is more adaptable to new platforms or demands.

Although the decision between CORBA/EJB and COM+ is easy to make it is unfortunate that there is little choice. However choosing between CORBA/EJB products may pose as much more of a challenge, but that is not the topic of this essay.

On a more general note, component technology as outlined in the Introduction has failed. The concept of a component being able to communicate with
another component regardless of location or platform just did not work. A COM+ component cannot talk to a CORBA/EJB object without a lot of hassles.

Why did it not work then? There are two reasons I believe, one is technical and the other non-technical. Firstly, the problem is a difficult one. This is not a simple platform independent communication problem such as TCP/IP (which incidently took decades to become universally dominant). There needs to be support for all the extra bells and whistles. It is simply not that easy. The OMG took the approach of writing a standard, and MS an implementation. The implementation approach cannot work unless it is open so it can be ported to other platforms. The standard approach is difficult to pull off when the problem is hard. Who really can read thousands of pages of specification and produce a product that complies with it. Obviously most of the vendors of CORBA/EJB could not since they are not interoperable. OMG needed to produce an open reference implementation along with their standard.

The second problem is greed. All the OMG members drafting the standard had their own interests at heart and bent the standard this way and that trying to make it fit their product. This combined with the difficulty of the problem at hand resulted in an enormous and loose standard. MS also did not want it to succeed, they wanted their technology to dominate. CORBA did try and bridge the gap by defining interoperability standards with COM, but this did not take off in the products.

It was the combination of these two issues, one technical and one non-technical, that was the main downfall of component technology.

Despite the failure of the original goal, component technology has helped software development. The only area where it really failed was interoperability.
So in a controlled, fairly homogeneous environment where interoperability does not matter, a distributed application can take advantage of all the other benefits component technology has to offer. Components can be written in different languages, moved from machine to machine without rewriting software and so on.

Even in one machine, being able to do interprocess communication in a relatively simple, robust and extensible way is a boon. Libraries do not have to be chosen based upon the language of implementation, rather their features can be the determining factor. Debugging is aided because parts of the application run in different address spaces.

It is sad that such lofty goals often fail to come to fruition. But without such grand plans we probably would not get anywhere at all.
Bibliography


